18-600 Recitation #11
Malloc Lab

November 7th, 2017

Important Notes about Malloc Lab

- **Malloc lab has been updated from previous years**
- **Supports a full 64 bit address space rather than 32 bit**
- **Encourages a new programming style**
  - Use structures instead of macros
  - Study the baseline implementation of implicit allocator to get a better idea
- **Divided into two phases:**
  - Checkpoint 1: Due date: 11/17
  - Final: Due date: 11/27
- **Try to finish Cache Lab by Thursday; it will help with Malloc and during a much-needed Thanksgiving break!**
- **Get a correct, reasonably performing malloc by checkpoint**
- **Optimize malloc by final submission**

Pointers: casting, arithmetic, and dereferencing

Pointer casting

- **Cast from**
  - <type_a>* to <type_b>*
    - Gives back the same value
    - Changes the behavior that will happen when dereferenced
  - <type_a>* to integer/ unsigned int / long
    - Pointers are really just 8-byte numbers
    - Taking advantage of this is an important part of malloc lab
    - Be careful, though, as this can easily lead to errors
  - integer/ unsigned int to <type_a>*

Pointer arithmetic

- The expression `ptr + a` doesn't mean the same thing as it would if `ptr` were an integer.
- Example:

```
type_a* pointer = …;
(void *) pointer2 = (void *) (pointer + a);
```

- This is really computing:
  - `pointer2 = pointer + (a * sizeof(type_a))`
  - `lea (pointer, a, sizeof(type_a)), pointer2;`
- Pointer arithmetic on `void*` is undefined

Pointer arithmetic

- ```
  int * ptr = (int *)0x12341230;
  int * ptr2 = ptr + 1;
  ```

- ```
  char * ptr = (char *)0x12341230;
  char * ptr2 = ptr + 1;
  ```

- ```
  int * ptr = (int *)0x12341230;
  int * ptr2 = ((int *) (((char *) ptr) + 1));
  ```
  ```

Pointer arithmetic

- ```
  int * ptr = (int *)0x12341230;
  int * ptr2 = ptr + 1; //ptr2 is 0x12341234
  ```

- ```
  char * ptr = (char *)0x12341230;
  char * ptr2 = ptr + 1; //ptr2 is 0x12341231
  ```

- ```
  int * ptr = (int *)0x12341230;
  int * ptr2 = ((int *) (((char *) ptr) + 1));
  //ptr2 is 0x12341231
  ```
  ```

Pointer dereferencing

- **Basics**
  - It must be a POINTER type (or cast to one) at the time of dereference
  - Cannot dereference expressions with type `void*`
  - Dereferencing a `t*` evaluates to a value with type `t`

Pointer dereferencing

- What gets “returned?”

```
 int * ptr1 = (int *) malloc(sizeof(int));
*ptr1 = 0xdeadbeef;

 int val1 = *ptr1;
 int val2 = (int) *((char *) ptr1);
```

What are val1 and val2?

Pointer dereferencing

- What gets “returned?”

```
 int * ptr1 = (int *) malloc(sizeof(int));
*ptr1 = 0xdeadbeef;

 int val1 = *ptr1;
 int val2 = (int) *((char *) ptr1);
```

```
// val1 = 0xdeadbeef;
// val2 = 0xffffffef;
```

What happened??

Malloc basics

- **What is dynamic memory allocation?**

- **Terms you will need to know**
  - malloc/ calloc / realloc
  - free
  - sbrk
  - payload
  - fragmentation (internal vs. external)
  - coalescing
    - Bi-directional
    - Immediate vs. Deferred

Concept

- **Really, malloc only does three things:**

1. Organize all blocks and store information about them in a structured way.
2. Using the structure made in 1), choose an appropriate location to allocate new memory.
3. Update the structure made in 1) when the user frees a block of memory

Allocation Example

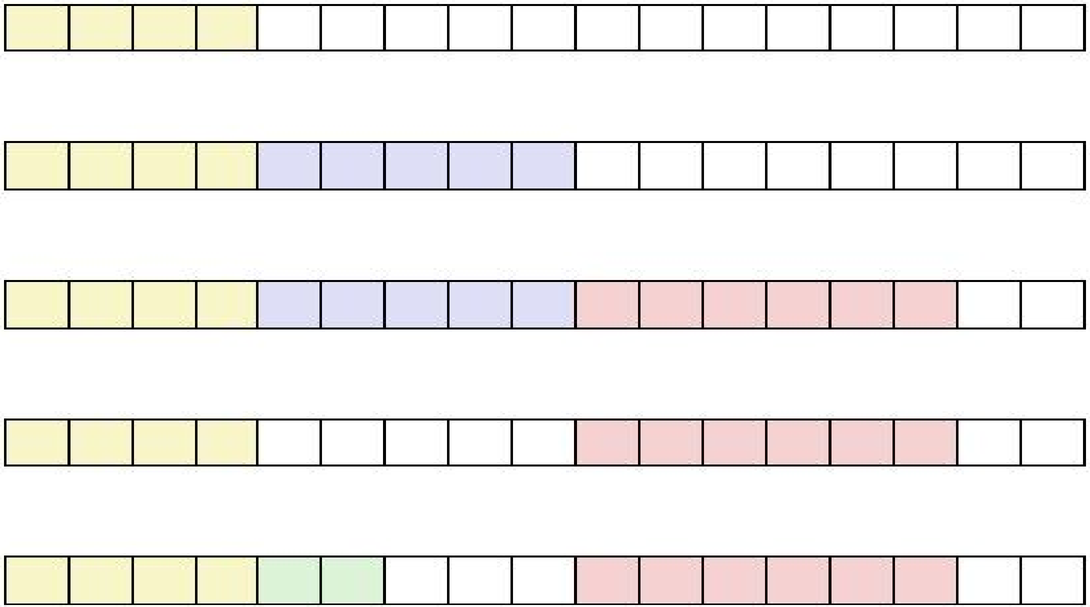
```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```



Fragmentation

- **Internal fragmentation**
  - Result of **payload** being smaller than block size.
  - `void * m1 = malloc(3); void * m2 = malloc(3);`
  - `m1, m2` both have to be aligned to 16 bytes…

- **External fragmentation**

External Fragmentation

- Occurs when there is enough aggregate heap memory, but no single free block is large enough

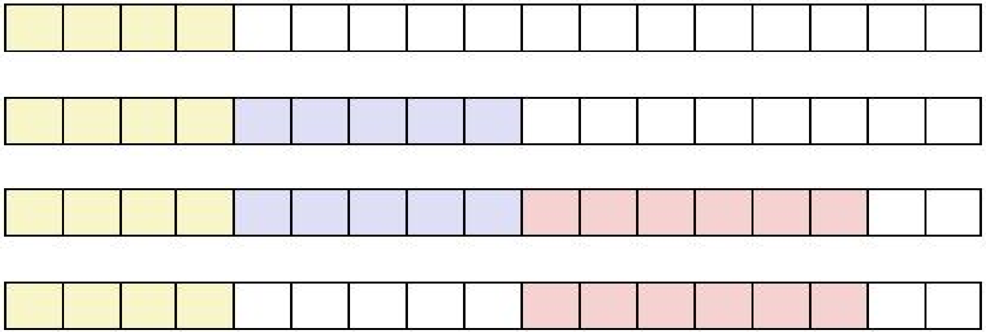
```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(6)
```



Oops! (what would happen now?)

- Depends on the pattern of future requests
  - Thus, difficult to measure

Goals

1. Run as fast as possible
2. Waste as little memory as possible

What kind of implementation to use?

- Implicit list, explicit list, segregated lists, binary tree methods …etc
- Can use specialized strategies depending on the size of allocations
- Adaptive algorithms are fine, though not necessary to get 100%.

What fit algorithm to use?

- Best fit: choose the smallest block that is big enough to fit the requested allocation size
- First fit / next fit: search linearly starting from some location, and pick the first block that fits.

  Which one's faster, and which one uses less memory?

Implicit List

- From the root, can traverse across blocks using headers which store the size of the block
- Can find a free block this way
- Can take a while to find a free block
  - How would you know when you have to call sbrk?

Explicit List

- **Improvement over implicit list**
- **From a root, keep track of all free blocks in a (doubly) linked list**
  - Remember a doubly linked list has pointers to next and previous
  - Optimization: using a singly linked list instead (how could we do this?)
- **When malloc is called, can now find a free block quickly**
  - What happens if the list is a bunch of small free blocks but we want a really big one?
  - How can we speed this up?

Segregated List

- **An optimization for explicit lists**
- **Can be thought of as multiple explicit lists**
  - What should we group by?
- **Grouped by size – let us quickly find a block of the size we want**
- **What size/number of buckets should we use?**
  - This is up to you to decide

Implementation Hurdles

- **How do we know where the blocks are?**
- **How do we know how big the blocks are?**
- **How do we know which blocks are free?**
- **Remember: can't buffer calls to malloc and free... must deal with them real-time.**
- **Remember: calls to `free` only takes a pointer, not a pointer and a size.**
- **Solution: <u>Need a data structure to store information on the "blocks"</u>**
  - Where do I keep this data structure?
  - We can't allocate a space for it, that's what we are writing!

The data structure

- **Requirements:**
  - The data structure needs to tell us where the blocks are, how big they are, and whether they're free
  - We need to be able to CHANGE the data structure during calls to malloc and free
  - We need to be able to find the **next free block** that is "a good fit for" a given payload
  - We need to be able to quickly mark a block as free/allocated
  - We need to be able to detect when we're out of blocks.
    - What do we do when we're out of blocks?

The data structure

- **Common types**
  - Implicit List
    - Root -> block1 -> block2 -> block3 -> …
  - Explicit List
    - Root -> free block 1 -> free block 2 -> free block 3 -> …
  - Segregated List
    - Small-malloc root -> free small block 1 -> free small block 2 -> …
    - Medium-malloc root -> free medium block 1 -> …
    - Large-malloc root -> free block chunk1 -> …

Playing with structures

- Consider the following structure, where a 'block' refers to an allocation unit
- Each block consists of some metadata (header) and the actual data (payload)

```
/* Basic declarations */

typedef uint64_t word_t;
static const size_t wsize = sizeof(word_t);

typedef struct block {
     word_t header;
     word_t alloc;
     char payload[0];
} block_t;
```

Why is this reasonable?

```
/* Basic declarations */

typedef uint64_t word_t;
static const size_t wsize = sizeof(word_t);

typedef struct block {
  // Header contains size + allocation flag
     word_t header;
     char payload[0];
} block_t;
```

Playing with structures

- **The contents of the header is populated as follows**

```
/* Pack size and allocation bit into single
word */

static word_t pack(size_t size, bool alloc) {

    return size | alloc;

}
```

```
/* Basic declarations */

typedef uint64_t word_t;
static const size_t wsize = sizeof(word_t);

typedef struct block {
  // Header contains size + allocation flag
     word_t header;
     char payload[0];
} block_t;
```

Playing with structures

- **How do we set the value in the header, given the block and values?**

```
/* Set fields in block header */

static void write_header(block_t *block,
                         size_t size, bool alloc) {

     block->header = pack(size, alloc);

}
```

```
/* Basic declarations */

typedef uint64_t word_t;
static const size_t wsize = sizeof(word_t);

typedef struct block {
  // Header contains size + allocation flag
     word_t header;
     char payload[0];
} block_t;
```

Playing with structures

- **How do we extract the value of the size, given the header?**
- **How do we extract the value of the size, given pointer to block?**

```
/* Extract size from header */

static size_t extract_size(word_t word) {

      return (word & ~(word_t) 0x7);

}

/* Get block size */

static size_t get_size(block_t *block) {

      return extract_size(block->header);
}
```

```
/* Basic declarations */

typedef uint64_t word_t;
static const size_t wsize = sizeof(word_t);

typedef struct block {
  // Header contains size + allocation flag
      word_t header;
      char payload[0];
} block_t;
```

Playing with structures

- **How do we write to the end of the block?**

```
/* Set fields in block footer */

static void write_footer(block_t *block,
                         size_t size,
                         bool alloc) {

    word_t *footerp = (word_t *)((block->payload) +
                  get_size(block) - 2*wsize);

    *footerp = pack(size, alloc);

}
```

```
/* Basic declarations */

typedef uint64_t word_t;
static const size_t wsize = sizeof(word_t);

typedef struct block {
  // Header contains size + allocation flag
     word_t header;
     char payload[0];
} block_t;
```

Playing with structures

- **How do we get to the start of the block, given the pointer to the payload?**

```
/* Locate start of block, given pointer to payload */

static block_t *payload_to_header(void *bp) {

     return (block_t *)(((char *)bp) -
                 offsetof(block_t, payload));

}
```

```
/* Basic declarations */

typedef uint64_t word_t;
static const size_t wsize = sizeof(word_t);

typedef struct block {
  // Header contains size + allocation flag
     word_t header;
     char payload[0];
} block_t;
```

GDB Practice

- Using GDB well in Malloc lab can save you **HOURS** of debugging time!
- Turn off gcc optimization before running GDB (-O0)
  - Don't forget to turn it back on (-O3) for the benchmark!

5 commands to remember:

1. backtrace
2. frame
3. disassemble
4. print <reg>
5. watch

Words of Wisdom

- Write a heap checker **first**. Please just do it and thank us later!
  Check for conditions that you know your heap should have.
- Printf <<<<< GDB
- Use version control, otherwise you'll regret it
- Don't feel bad about throwing away broken solutions!
- **Start early**, read the handout carefully.
- **Warnings:**
  - Most existing Malloc literature from the book has slightly different guidelines, they may be out of date